INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Structured Scheduling of Recurrence Equations

Tanguy Risset, Florent Dupont de Dinechin, and Sophie Robert

N ° 3282

THÈME 1

ISSN 0249-6399

Structured Scheduling of Recurrence Equations

Tanguy Risset, Florent Dupont de Dinechin, and Sophie Robert

Thème 1 — Réseaux et systèmes
Projet API

Rapport de recherche n3282 — — 41 pages

Abstract: We study scheduling of structured systems of recurrence equations. We first recall the formalism of structured systems of recurrence equations, then we explain how to implement a scheduling tool for non-structured systems of reccurence equations. We introduce the notion of structured schedule and propose criteria for determining whether a structured system of reccurence equations admits a structured schedule. We give an algorithm for its computation. Finaly we propose an algorithm for computing structured multi-dimensionnal schedule for systems which do not admit a structured linear schedule. With this last method, we derive a structured schedule for a structured system of recurrence equations computing the singular value decomposition of a matrix.

Key-words: cyclic scheduling, structured recurrence equations, regular parallelism, loop nest, Automatic synthesis methodology, vlsi

(Résumé : tsvp)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Ordonnancement structuré d'équations récurrentes

Résumé : Nous étudions l'ordonnancement de systèmes structurés d'équations récurrentes. Après un rappel du formalisme des équations récurrentes structurées, nous expliquons comment mener à bien l'implémentation d'un outil d'ordonnancement pour des systèmes non structurés d'équations réccurentes. Puis nous introduisons la notion d'ordonnancement structuré et nous proposons des critères permettant de déterminer si un système structuré d'équations récurrentes admet un ordonnancement structuré linéaire et nous proposons un algorithme pour le calculer. Enfin nous proposons un algorithme pour calculer un ordonnancement multidimensionnel structuré pour les systèmes n'admettant pas d'ordonnancement linéaire. Grâce à cette méthode nous calculons un ordonnancement structuré pour un système effectuant la décomposition en valeurs singulières d'une matrice.

Mots-clé : ordonnancement cyclique, équations récurrentes structurées, parallelisme régulier, nids de boucles, méthodologie de synthèse automatique, vlsi

Contents

1 Introduction

Recurrence equations where introduced as a formalism for describing computation by Karp, Miller and Winograd [1]. This formalism has been widely studied in the field of parallelization [2, 3, 4, 5] and synthesis of regular arrays [6, 7, 8, 9]. Works on the formalism of recurrence equations lead to various languages like Lucid [10], Lustre [11], Signal [12], Crystal [13], Pei [14] and Alpha [15]. Works on the scheduling of recurrence equations lead to important results [16, 17, 8, 3, 18]. In this paper, we study systems of recurrence equations (SAREs) withx the formalism of the Alpha language. Recently, Alpha was extended to include program structuring [19]. This allowed to write and manipulate big SAREs specifications in a structured form, but raised several questions. For instance, given a SARE which is used in different contexts, which schedule should we give to this system? A structured scheduling should provide a single scheduling for all the different uses, this is not always possible and scheduling structured SARE cannot be directly deduced form the scheduling techniques existing for unstructured SAREs.

This paper is concerned by the scheduling of structured SAREs, we first present practical results on the scheduling of unstructured SAREs, then we propose a methodology for scheduling structured SAREs. All our example will be specified in Alpha, hence we first explain in depth the Alpha formalism in section 2. Then we detail the structuring of Alpha in section 3, in particular, we introduce the notion of structured dependence graph as an extension of the classical dependence graph (or reduced dependence graph) used in systolic synthesis. The structuring of Alpha has already been presented in [19, 20] but its understanding is fundamental for the rest of the paper. In section 4, we explain how to build an efficient schedule tool for SAREs. As the foundations of the scheduling method has been presented [3, 18, 17, 9], this part consists in results from practical scheduling and high level built-in options that should provide a schedule tool for VLSI synthesis. Section 5 deals with structured linear scheduling and isolates some cases of structured SAREs where a structured linear scheduling can be found. Then a method is depicted in section 6 for finding multidimensional structured scheduling when linear scheduling does not exists. Finally we apply the method on a structured SARE which represent the computation of the singular value decomposition of a matrix.

2 Simple SAREs in Alpha

This section gives a definition of the syntax and semantics of the ALPHA language. ALPHA being basically a convenient notation for the formalism of polyhedral affine recurrence equations, we first define polyhedra and space variables, the basics of this formalism. Then we introduce in 2.2 the ALPHA expressions and give their compositional semantics. Subsection 2.3 defines the syntax and semantics of an affine recurrence equation, and subsection 2.4 defines a system of such AREs.

The first parts of this section are illustrated by the very classical example of an ALPHA matrix-vector product (Prog. 1).

```
1   system matvect (M : {i,j | 1<=i<=10; 1<=j<=10} of real;
2                   V : {j | 1<=j<=10} of real)
3          returns (R : {i | 1<=i<=10} of real);
4   var
5     C : {i,j | 1<=i<=10; 0<=j<=10} of real;
6   let
7     C = case
8           {i,j | j=0} : 0.(i,j->);
9           {i,j | j>0} : C.(i,j->i,j-1) + M * V.(i,j->j);
10        esac;
11    R = C.(i->i,10);
12  tel;
```

Program **1:** Matrix-vector product

In the description of the syntax of the language, we use the following conventions :

*phrase**	denotes zero or more repetitions of *phrase*.
phrase1 | *phrase2*	denotes alternation, either *phrase1* or *phrase2*.
[...]	denotes optional phrase.
(...)	denotes syntactic grouping.
`courier`	denotes a terminal.
Italic	denotes a non-terminal.

2.1 Basics

The dominant data structure in ALPHA is the *polyhedral data array*. Such an array is a function of some integer vector space $\mathbb{Z}^n$ to a scalar value space (integer, boolean or real). The domain of this function is a convex polyhedron of $\mathbb{Z}^n$. Intuitively, this data structure enables to represent vectors, matrices, any n-dimensional array, but also more complex data shapes such as triangular matrices (e.g. `{i,j| 1<=i,j<=10; j<=i}`). Infinite domains are also allowed to represent, for example, infinite data-streams (e.g. `{t| t>=0}`).

ALPHA variables hold such polyhedral data arrays: thus a variable v is declared by specifying its domain and the type of its values. For example, in Prog. 1, line 5 declares a variable named `C` which is a square 10×10 matrix.

The general syntax for a polyhedron is the following:

Domain :: `{` *IndexList* `|` *ConstraintList* `}`

Actually the language allows *finite unions* of convex polyhedra: the set *DOM* of all the finite unions of integral convex polyhedra is closed under intersection, union, set difference,

preimage by an affine function, and convex hull of the image by an affine function [21]. These good properties allow the formal transformation of ALPHA variables.

ALPHA is also strongly typed with respect to the scalar type of the values held in polyhedral data arrays (integer, real or boolean). We will not detail this scalar typing, which is based on very basic and classical type inference techniques.

2.2 Expressions

ALPHA expressions are built out of the previous variables, and constants, the latter being also considered as polyhedral data arrays of dimension zero. An expression also holds a polyhedral data array, which is defined compositionally from those of its subexpressions.

The general syntax of an expression is:

Expression ::

Variable | *Constant*	(Terminals)
| *UnOp Expression*	(Pointwise unary)
| *Expression BinOp Expression*	(Pointwise binary)
| `if` *Expression* `then` *Expression* `else` *Expression*	(Pointwise `if`)
| *Domain* : *Expression*	(Restriction)
| `case` *ExpressionList* `esac`	(Disjunction)
| *Expression* . *AffineFn*	(Affine dependency)
| `reduce(`*BinOp* , *Expression* , *AffineFn* `)`	(Reduction)
| `(` *Expression* `)`	

There are two kinds of operators: *pointwise* computation operators transform the values, and *spatial* operators manipulate the domains of the data arrays. In addition, the *reduction* operator is both computational and spatial.

Pointwise operators A pointwise operator describes a computation applied to all the points of a data array. For example, if E_1 and E_2 are two expressions of the same dimension, then $E_1 + E_2$ is an expression of the same dimension, defined anywhere where both E_1 and E_2 are defined, and whose value at each point is the sum of the values of E_1 and E_2 at this point.

More formally, the domain of $E_1 + E_2$ is the intersection of the domains of E_1 and E_2, and its value function is the sum of those of E_1 and E_2.

Other pointwise operators include most unary and binary arithmetic and logical operators, comparison operators, plus the ternary `if then else` operator.

Spatial operators The *restriction* of the expression E to the domain D is noted $D : E$. It is an expression whose values are those of E, but whose domain is the intersection of the domain of E with D.

The *case* operator allows the piecewise definition of an expression by several subexpressions defined over disjoint domains. For example the expression `case` $E_1; E_2;$ `esac` is an expression whose domain is the union of the domains of E_1 and E_2. Its values are defined as those of E_1 over the domain of E_1, and those of E_2 over the domain of E_2. If both values are defined on some point, this expression is undefined. To avoid this situation, the restriction operator is usually needed to restrict E_1 and E_2 properly. See Prog. 1 for an example.

Finally, the *dependence* operator establishes an affine mapping from the points of a domain to the points of another domain. Typical affine dependencies are translations or value broadcast. If we call f the affine function, $E.f$ is an expression whose value at a point $\mathbf{z}$ is the value of E at the point $f(\mathbf{z})$, and whose domain is therefore the preimage by f of the domain of E.

The syntax of affine functions is :

AffineFn :: `(` *IndexList* `->` *IndexExpList* `)`

where the index expressions relate the indices in the index list.

Notice that a constant is a data array of dimension zero, and therefore usually needs a dependence function to be turned to a data array of greater dimension. For example, if `A` is a matrix which we want to add 42 to each element of `A`, then `A + 42` is incorrect, for it is the pointwise sum of an object of dimension 2 and an object of dimension 0. The correct way to express this is to build a two dimensional array of 42s using a dependency operator, as in `A + 42.(i,j->)`. See also line 8 of Prog. 1.

The reduction operator This operator allows to write high-level expression such as the summation of the introduction. Let us take another example, our matrix-vector product of Prog. 1. It could be expressed in a more abstract manner as $R_i = \sum_{j=1}^{N} M_{ij} V_j$. This sum is expressed in ALPHA as the following line, which could replace lines 7 to 11:

```
R = reduce( +, (i,j->i), M*V.(i,j->j) );
```

The reduce operator needs a binary associative and commutative operator (here +), an ALPHA expression (here `M*V.(i,j->j)`) of domain D (here a matrix) and an affine projection f (here `(i,j->i)`) specifying in which direction(s) of D the reduction is to take place. The domain of this `reduce` expression is the image of D by this projection, and the value of each point of this image is the combination of all its antecedents by f in D.

An automatic program transformation exists that expands the line above into lines 7-11 of Prog. 1.

2.3 Equations

An ALPHA program is basically a system of equations, each equation having a variable as left-hand side and an expression as right-hand side.

The usual syntax of an equation is therefore:

Equation :: *Identifier* = *Expression* ;

It is important to distinguish such an equation from an affectation: the ALPHA equation expresses an identity between both sides. In particular, to be valid, an equation must define the whole of the variable, that is, the domain of the expression must at least cover the domain declared for the variable.

Array notation In most cases, the dependence operator may be rendered with an easier-to-read array notation [22]. This notation is nothing but a simpler syntax for an equation which has several index declarations within domains or affine functions on its right-hand side (e.g. `i,j` in lines 8-9 of Prog. 1). If the dimensions are coherent, then these declarations are similar (the names of the indices may differ, but their number is the same), and the equation may be simplified by putting them on the left-hand side. An example of this is Prog. 2, which is the core of Prog. 1 written in array notation.

```
7    C[i,j] = case
8                {| j=0} : 0[];
9                {| j>0} : C[i,j-1] + M[i,j] * V[j];
10               esac;
11    R[i] = C[i,10];
```

Program **2:** Array notation for the matrix-vector product

This array syntax is no longer compositional, and thus may not be used in all cases, but it will be adequate for the rest of this paper. The reader, however, should keep in mind that square bracket are nothing but a convenient notation for affine dependency functions.

2.4 Systems

An ALPHA system is a set of mutually recursive equations. All the variables are declared in the header. There are three classes of variables. Input variables only appear on the RHS of the equations: their domain is declared in the header of the system, but they do not have an equation defining them. Output variables are returned by the system. Local variable are auxiliary variables. The syntax of a system is thus:

SystemDecl ::
`system` *Name* `(` *InputDeclList* `)`
`returns` `(` *OutputDeclList* `)` `;`
[`var` *LocalDeclList* `;` *]*
`let`
EquationList
`tel`

Each declaration list contains declarations of variables as described in 2.1.

2.5 Size parameters

Prog. 1 is a matrix-vector product of fixed size 10. This obviously prevents it from being used in a more complex application in an useful way, for an other application will need matrix-vector products of different size, the size even varying within the application. What is needed is therefore a more *generic* system, describing a matrix-vector product of arbitrary size, say N. The actual size should be an implementation detail.

The polyhedral SARE model allows for a simple but powerful parameter scheme: all we need is add an index `N` to each polyhedron of the system, and consider this index as a size parameter. For example the domain `{i,j | 1<=i,j<=10}` becomes `{i,j,N | 1<=i,j<=N}`. This index also needs adding in affine functions to replace the 10s there. Thus the dependency `(i->i,10)` becomes `(i,N->i,N,N)`. The `N` on the right-hand side is needed for dimension consistency. It is easy to show that the modified system is a valid ALPHA program, and that it can be analyzed and transformed as well (we will come back on this later).

Now we have added a "`,N`" to all the index lists of polyhedra, and to the left-hand and right-hand sides of all the affine functions. There is a lot of redundancy there, because this parameter index actually stands for a *global* constant. Therefore it makes sense to declare it only once, in the beginning of the system, and then omit its declaration in the rest of the system. Besides, when declaring it, we may as well declare a domain of values permitted for this parameter, as a usual polyhedral domain.

```
1   system matvect : {N| N>1}
2                  (M : {i,j | 1<=i,j<=N} of real;
3                   V : {j | 1<=j<=N} of real)
4          returns (R : {i | 1<=i<=N} of real);
5   var
6     C : {i,j | 1<=i,j<=N} of real;
7   let
8     C = case
9           {i,j | j=0} : 0.(i,j->);
10          {i,j | j>0} : C.(i,j->i,j-1) + M * V.(i,j->j);
11        esac;
12    R = C.(i->i,N);
13  tel;
```

Program **3:** Parameterized matrix-vector product

This defines the syntax for parameterized ALPHA systems, exemplified by Prog. 3. More generally, the parameter domain may be any ALPHA domain, in particular there is no limit on the number of parameters, and it is possible to express any affine constraint between the parameters. For example, `{M,N | M<2N}` is a valid parameter domain. Parameters may then appear anywhere in the system where indices are allowed, that is, in domains (see lines 2-4 of Prog. 1) as well as in affine functions (see line 12). This parameterized syntax

obviously retains all the properties of the language, since it is always possible to write the parameterized system as a non-parameterized one.

With respect to program analysis and transformation, however, parameterization is more than simply syntactic: parameters are syntactically indices, but their semantic is that of a constant in a system. For example, the parameterized affine function `(i,j->i+N, j+N)` is a translation and should be considered as such in the uniformization process, although its closed affine form `(i,j,N->i+N, j+N, N)` is not a translation.

3 Structured SAREs in Alpha

The issue of structuring a complex algorithm into a hierarchy of SAREs is more complex than it seems. Obviously, it is partly addressed by the decomposition of the problem into equations: it is always possible to break an equation into two simpler equations, using an auxiliary variable to hold a subexpression of the initial expression. The reverse operation, replacing a variable appearing in an expression with its definition, is also always possible (these operations are similar to the β-conversion in the lambda-calculus).

3.1 The need for higher-order program structures

Structuring using equations, however, basically remains first-order structuring (in the usual functional meaning), and is thus limited. These limits appear more clearly on the following example: suppose we want to write in Alpha an algorithm for a signal-processing application, which involves time-varying matrices. Such matrices are represented in Alpha as three-dimensional data arrays (two matrix dimensions, and one – possibly unbounded – time dimension) as represented on Fig. 1.

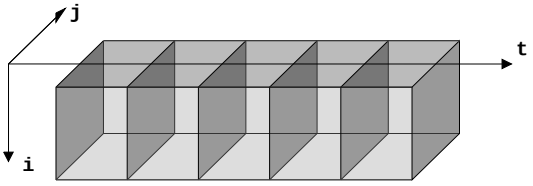


Figure 1: Linear collection of bi-dimensional matrices

Now let us try to re-use the equations of the matrix-vector product to operate on these time-varying matrices. Obviously it is not possible in a straightforward manner, for the inputs don't have the proper dimension. For example we will need to rewrite the whole of the equation defining `C` to add one dimension to the domains and the affine functions, as shown by Prog. 4.

Structuring with variable will never be adequate when such a *dimension extension* is needed, which is a very common case: it corresponds to a procedure or function call within a loop nest in imperative languages, or to a `map` structure in functional languages.

```
8  C = case
9       {i,j,t | j=0}: 0.(i,j,t->);
10      {i,j,t | j>0}: C.(i,j,t->i,j-1,t) + M * V.(i,j,t->j,t);
11     esac;
```

Program **4:** Using the matrix-vector product on time-varying matrices

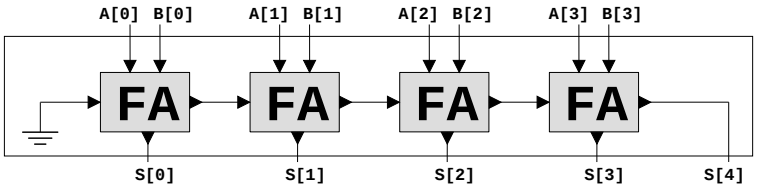


Figure 2: An adder

3.2 Introductory example

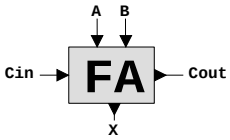
As another, more concrete example, let us try and write ALPHA SAREs for the addition and multiplication of two integers (or fixed-point numbers) written in binary notation. These numbers will be coded as arrays of booleans in ALPHA.

The base block of these operation is the *full adder* function which takes three binary inputs A, B and C_{in} and expresses their sum on two bits X and C_{out}:

$$A + B + C_{in} \;=\; X + 2C_{out}$$

Prog. 5 is a full adder function written in ALPHA. In this system, the domains of all the variables are $\mathbb{Z}^0$ (one point).

```
system FullAdder (A,B,Cin : boolean)
         returns (X,Cout : boolean);
let
  X= A xor B xor Cin;
  Cout= (A and B) or (A and Cin) or (B and Cin);
tel;
```



Program **5:** Full Adder system

An adder is classically described as a sequence of full adders with carry propagation, (hence the names C_{in} and C_{out}, for "carry") as shown on Fig. 2.

In order to re-use the system given as Prog. 5, we need to keep the same equations, but to change the domains of the variables so that they become arrays of bits:

```
A,B,Cin,Cout,X : {b| 0<=b<W} of boolean;
```

The structure construct `use` in ALPHA allows a system to be used by another one with an *extension of the dimensionality* of the subsystem. This dimension extension is expressed as an ALPHA domain, in our example `{b| 0<=b<W}`. An example of this feature is Prog. 6.

```
1     system Plus: {W|W>1} (A,B: {b| 0<=b<W} of boolean)
2                  returns (S : {b| 0<=b<=W} of boolean);
3     var
4       Cin, Cout, X : {b| 0<=b<W} of boolean;
5     let
6       Cin[b] =
7         case
8           {| b=0} : 0[];
9           {| b>0} : Cout[b-1];
10        esac;
11      use {b| 0<=b<W} FullAdder[] (A,B,Cin) returns(X,Cout);
12      S[b] =
13        case
14          {| b<W} : X;
15          {| b=W} : Cout[W-1];
16        esac;
17    tel;
```

Program **6:** Addition using subsystem `FullAdder`

In this system, the line 11 reads as follows: "Use a *collection* of instances of the subsystem `FullAdder`. This collection has the shape of the extension domain `{b| 0<=b<W}` and is thus indexed by index `b`. Let the inputs of the `b`-th instance be the variables `A`, `B` and `Cin` at point `b`, and similarly let the outputs of this collection of instances be the variables `X` and `Cout` at point `b`."

The lines 6-10 describe the carry propagation, and lines 12-16 define the output of this binary adder.

This `use` construct with extension domain may be viewed as a `map` operator in more conventional functional languages. In ALPHA, however, the extension domain may be any ALPHA domain, of arbitrary dimension and arbitrary polyhedral shape.

3.3 Inputs and outputs

In Prog. 6, the actual inputs given to the system (`A`, `B` and `Cin`) are all variables. In general, however, they may be any expression: their semantic is that of a right-hand side of expression, and the link between actual and formal inputs is a virtual *equation* (in the usual ARE sense) [19].

Actual outputs, on the other hand, have a left-hand side semantics, and so they have to be variables only.

Here is the general syntax of a `use` construct. It should be noted that this construct appears at the syntactic level of an equation, for its intuitive semantics is that of a set of equations (those of the subsystem).

Equation : :
    Identifier = *Expression* ;
  | `use` [*ExtensionDomain*] *Identifier*
        [[*ParamAssignation*]]
        (*ExpressionList*)
    `returns` (*IdentifierList*) ;

3.4 Giving values to the subsystem's size parameters

In Prog. 6 the subsystem `FullAdder` has no parameter. In the general case, the parameters of the subsystem need to be given a value, which may be any affine function of the parameters of the caller and the extension indices.

This feature is best explained using some examples. Consider the binary multiplication algorithm, which is very similar to the decimal usual algorithm performed "by hand". Fig. 3 shows that such a multiplication is basically a collection of additions.

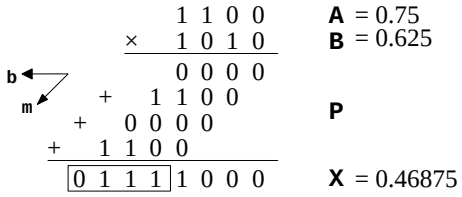


Figure 3: Product of two fixed point reals in binary representation

Prog. 7 is the ALPHA specification of Fig. 3. Line 8 performs the binary product of all the bits of the first operand by each bit of the second. Line 10 describes a linear collection of additions (described in program 6), indexed by `m` which is the row index in Fig. 3. Lines 12-17 link the result of one additions to the input of the following.

Here the parameter assignation `W` just equates the bit size parameter of the subsystem `Plus` and that of the multiplication. In the general case, however, this parameter assignation may be any affine function of the caller's parameters and also the extension indices. An example where the "natural" structuring of an algorithm makes use of a parameter assignment depending on the extension indices is the Gaussian elimination [23].

```
1       system Times: {W|W>2} (A,B: {b| 0<=b<W} of boolean)
2                     returns (X  : {b| 0<=b<W} of boolean);
3       var
4         P : {b,m| 0<=b,m<W } of boolean;
5         Si : {b,m| 0<=b<W; 0<m<W } of boolean;
6         So : {b,m| 0<=b<=W; 0<m<W } of boolean;
7       let
8         P[b,m] = A[b] and B[m];
9
10        use {m| 0<m<W} Plus[W] (Si,P) returns (So);
11
12        Si[b,m] =
13          case
14            {| m=1} : P[b,m-1];
15            {| m>1} : So[b+1,m-1];
16          esac;
17        X[b] = So[b+1,W-1];
18      tel;
```

Program **7:** Binary multiplication in ALPHA

3.5 A substitution semantics

The semantics of a subsystem `use` is more subtle than it seems. The main difference with a more conventional function or procedure call is that the `use` expresses a set of computations which *is not monolithic*: line 10 of Prog. 7 hides a two-dimensional collection of instances of full-adder functions without implying any order on the evaluation or execution of these instances. In particular, all the inputs need not being defined on the whole of their domains for the `use` to have a semantic. In this program it wouldn't be possible, since the input `Si` is defined as a function of the output `So`. From this point of view, the `use` is to the procedure call what the recurrence equation is to the affectation.

For this reason, it is difficult to give a compositional semantics to the `use` (the problem is addressed in [20]). It is much easier and intuitive to define this semantics by substitution: a program containing a `use` is equivalent to a program in which this `use` has been replaced with the equations of the subsystem, plus equations for actual/formal input/output linking. These equations are usually deeply modified because of dimension extension, but these modifications are based on the usual and well-defined domain computations. This substitution semantics is detailed in [19] and is referred in this paper as the *inlining* process. Given an Alpha structured system, the corresponding *inlined* system is the system in which each `use` has been substituted by the equation of the system used (and recursively).

3.6 Dependence graph for structured SAREs

Given an Alpha system (a system without a system use, or any kind of loop nest without procedure calls), the data dependency information is usually represented in the form of a dependence graph (or reduced dependence graph.

Definition 3.1 *(*dependence graph*) The dependence graph of an Alpha system (without system call) is a graph $G = (V, E)$ whose vertices V represent Alpha variables and edges E represent the dependencies between variables. Vertices are labeled by the variables names and each edge is labeled by the dependency function and the domain on which it applies.*

We need to extend the notion of dependence graph to systems containing system use. However, such a structured dependence graph should only contain informations available in the caller. Given a structured Alpha system (a system containing a system use), it is not possible to express the exact dependencies between the actual inputs and the actual outputs of the system called except if we analyze the dependencies in the body of this system called. Hence in that case, we provide systematically a dependence between each input and each output. This edge will be labeled by: the name of the subsystem, the domain on which this subsystem is applied (i.e. the extension domain) and the parameter assignement function (these notions have been explained in section 3). An example of structured dependence graph is given in section 5.

Definition 3.2 *(*structured dependence graph, SDG*) The dependence graph of a structured Alpha system (with system use) is a graph $G = (V, E)$ whose vertices V are labeled by the variables of the system and edges E represent:*

- *dependencies between variables, these edges are labeled by the dependency function and the domain on which it applies;*
- *dependencies between actual input and actual output of subsystem uses. These edges are labeled by the name of the system, the extension domain and the parameter assignement function.*

4 Experimenting the scheduling of recurrence equations

The problem of finding scheduling for algorithms expressed in terms of recurrence equation has been widely studied from the theoretical points of view. However, practical use of cyclic scheduling has still to be studied in depth. In this section we describe how to obtain a flexible recurrence equation scheduling tool for VLSI design. Experience comes from practical scheduling of recurrence equations for VLSI synthesis (see for instance [24, 25, 26])

Scheduling a SARE intends to give an execution date to each computations specified by the system. As there are usually a very large number of such computations, we look for a closed form to express the schedule of a possibly infinite number of computations in a finite manner. The formalism of recurrence equations naturally lead to the notion of *linear*

scheduling: the computation date of a variable `C` at index point `I` is expressed by a linear function of `I`: $T_C(I) = \tau_c \cdot I + \alpha_c$. From now on, we will use the following convention: T for schedule function, τ for the vector of coefficients (τ is often called the schedule vector) and α for the constant coefficient. In this paper we only consider the case where the schedule vector τ does not depend on the index I, hence we talk of *cyclic scheduling* (a generalization could extend the results presented here to the case where the domain of `C` is splitted into a finite number of sub-domains, the schedule vectors being constant on each sub-domains). This linear function T gives a partial order of execution and can be interpreted as the value of a global clock.

For example, consider the matrix vector product of program 3 (page 9), if we suppose that inputs `M` and `V` are available at time 0, one could propose the following schedule function for C:

$$T_C(i,j) = j \quad .$$

This schedule function is valid because all the dependencies of the program are respected (`C[i,j]` is computed after `C[i,j-1]` upon which it depends). It needs the assumption that one *clock cycle* is enough to compute one multiplication-addition and that we have enough processors to compute all the `C[i,j]`, $1 \leq i \leq N$ together.

As all SAREs do not admit a linear schedule, multi-dimensional schedules will be introduced in section 6. In this section, we first briefly recall the kernel principle for computing cyclic scheduling then we explain how to build a flexible schedule tool for SAREs.

4.1 Implementation of the kernel algorithm

The main difficulty of cyclic scheduling lies in the fact that we wish to express scheduling for large (or infinite) iteration spaces. This problem brought together two research domains, systolic synthesis [7, 8, 9] and automatic parallelization [3, 18, 4, 5, 27] (because loop nests computations can be modeled by recurrence equations). A first scheduling technique for uniform recurrences was proposed by Karp, Miller and Winograd [1], then progress were made to extend this method to affine recurrences [3, 17, 9]. All these methods use some form of the duality theorem. For instance the method depicted in [3] uses the affine form of Farkas Lemma, the one of [17] uses the duality theorem of linear programming.

We have chosen an implementation based on the method explained in [3, 18]. Starting from a recurrence equation specification expressed in Alpha, we express all the constraints that the timing vectors must check. For this, the basic principle is to expressed all the constraints in the following form:

$$F(x) \geq 0 \qquad \forall x \text{ such that } \quad Ax + b \geq 0 \tag{1}$$

where F is an affine function. The use of the Farkas lemma allows condition (1) to be transformed into the resolution of a system involving only F, A and b, hence independently of the size of the convex domain containing x. Applying Farkas lemma to all the constraints that the scheduling vectors must respect yields a big linear programming problem, the resolution of which provides the timing vectors. In our implementation, algorithmic treatments

are made with Mathematica, and the resolution of the linear programming problem is done with the PIP software [28]. The resolution can be done in rational mode. However, due to the design methodology following the scheduling process it is, in practice, much more convenient to consider the problem as an integer linear programming problem. The structure of the underlying problem is such that the execution time to solve the integer linear problem is roughly equal to the execution time needed to solve the same problem in rational. To our knowledge, this property remains to be proved.

Experiments show that, even if the implementation is not optimized, the execution time needed to find a schedule is not a real problem for small program because scheduling is a process which is done quite parsimoniously. For instance, with our implementation, scheduling system of program 3 (page 9), (which has 3 `case` branches) would take 15 seconds. For bigger program, the inner complexity of the linear program generated is a real problem. In our implementation, the biggest linear program generated was from a SVD specification (80 `case` branches) and was composed of 2000 variables and 4000 constraints (the resolution could finally be completed in 3 hours). For such big examples, it is very important to optimize the LP and reduce the number of variable, this can be done by:

- providing optional simplification when building the linear programming problem. For instance, merging union of convex polyhedra into the convex hull of the union or providing a special procedure when the program is uniform. In our implementation, such optimizations can divide the execution time by 5 without affecting to much the quality of the result.
- using an optimized LP-solver which uses a sparse representation of constraints (most of the coefficients of the constraints are zero). However, this solver should, as PIP does, select a lexicographic minimum of the vector of variables instead of only minimizing a linear form upon the variables because it provides a very useful way to choose between different valid solutions.

4.2 Implementation of a schedule tool

For a given Alpha program, there are many possible schedules. The choice of the schedule can be influenced by the objective function to be minimized. In [3, 18], two important cases are explained: minimizing the total execution time, and having bounded delays on dependencies. A lot of research have been made for finding the fastest schedules, however, we believe that this is not the most important issue when targeting VLSI system. Indeed, this objective is more or less always obtained when the coefficients of the timing function are minimized (provided that the domains are contained in the positive orthant). It is much more important to give the user the opportunity to select between different schedules, this can be done by three important means.

1. Selecting different orders for the coefficients to be minimized lexicographically. As PIP minimizes lexicographically all the variables of the LP, changing the order of the

variables affects the result. For instance, it is often more useful to minimize the coefficients of the parameters before the coefficients of the regular indices, this prevents *embarrassingly parallel* solutions without affecting the overall execution time (see section 4.3). This mechanism allows constants and parameters (which are often *large* constants) to be treated differently, this is one of the reason why it is much more flexible to schedule a parameterized program than the same program with the parameter having been set to a particular value.

2. Allowing to add arbitrary constraints. For instance, it may happen, and this is actually very often the case, that the inputs of the system are not all available together, but are computed, with another system, at certain dates. For that, the user should be able to indicate: "input `In[i,j]` is not available before date $T_{In}(i,j)$". In general, conditions to express may be more complicated, a good schedule tool should be able to use Farkas lemma dynamically to take into account additional constraints of the form of (1).

3. Allowing the user to decide the durations of each computations. In VLSI synthesis, the clock cycle is dependent of the longest path without register on it. Deciding that one operator will last one top is exactly equivalent to decide that there is a register after the operator. This is a designer choice and should be easily changeable by the user, hence the schedule tool should provide a simple mechanism for setting each duration of operation individually.

   Of course, this is a fastidious work, some default setting can be proposed. One natural strategy is to set the duration to one top by equation, from practical experiments, the strategy the closest to an ideal duration is to count 0 top for the operators, and one top for the non identical dependencies. This expresses the fact that the clock cycle of a VLSI synchronous circuit is determined by the delay of its critical path, which is the longest path between two registers. In the usual VLSI interpretation of a SARE, it is the non identical dependencies which are interpreted as registers, hence the choice of counting one clock top for such a dependency, and 0 for the operators which will be on a path between two registers. This leads, of course, to unbalanced path delays, which the designer may later improve by adding dependencies to split the critical path, or by using classical retiming techniques. However, it seems more realistic to have a more precise duration strategy which takes into account the operators used in the equations.

4. Providing *pre-defined* types of scheduling. For instance, when one manipulates a uniform SARE, it is often mandatory to search a schedule with the same linear part for all the variables of the program (otherwise the regularity of uniformity will be lost). But what is less obvious is that this unique linear part constraints should not apply on the coefficients of the parameters (`N`) because they should be considered as constants. From our experience in systolic synthesis the schedule for uniform programs should have such characteristics and should be available for the user.

In addition to these features, a good schedule tool should provide facilities to be used for the computation of multi-dimensional schedules. This include in particular a way to restrict the schedule to some variables and/or to some dependencies, as will be mentioned in section 6.

4.3 Example of flexible scheduling

Consider the $N \times N$ matrix-vector product of program refmatvect-param (page 9), a natural schedule would be:

$$T_C(i,j) = j \quad T_R(i) = N + 1$$

but, if the input `M[i,j]` is available at step i, the schedule obtained after adding this constraints could be:

$$T_C(i,j) = j + i \quad T_R(i) = N + i + 1 \quad . \tag{2}$$

Another valid solution would be:

$$T_C(i,j) = j + N \quad T_R(i) = 2N + 1 \quad . \tag{3}$$

The total execution time of (3) is the same, but this schedule imposes us to keep the coefficients of `M` in a memory while schedule (2) used each coefficient in a pipeline way as in a systolic architecture [29]. The difference between the two schedules comes from the fact that the second is obtained by minimizing the coefficients of the indices (i,j) before the coefficient of the parameter (N). Finally, note that the last equation of program 3 leads to a register in schedule (2) for each coefficient of `R`. If one wants to get rid of this register (and consider equation `R[i]=C[i,N]` as a simple connection), one simply indicates that this last equation have a duration of 0 and we obtain the following schedule:

$$T_C(i,j) = j + i \quad T_R(i) = N + i \quad .$$

5 Structured linear scheduling

Obtaining linear schedules for small Alpha programs is now easy. The structuring process described in section 3, allows to write big algorithms in a structured manner. Providing a schedule for a structured system of recurrence equation is not obvious, one can inline the structured program in order to obtain a single flat Alpha program and schedule it. But this method has several drawbacks: it looses the information of the structure of the program, it leads to huge linear programs which are often impossible to solve.

Another possibility is to look for a *structured scheduling*. We say that a schedule of an Alpha system `l1` is *structured* if any use of a given system `s1` can be scheduled with the same schedule (which is in fact the schedule of this system `s1`). One motivation could be, for instance, the use of library elements that has already been implemented. This approach corresponds to a divide and conquer strategy for scheduling and has many advantages: it permits to take into account the structure of the recurrence equation specification, to reuse

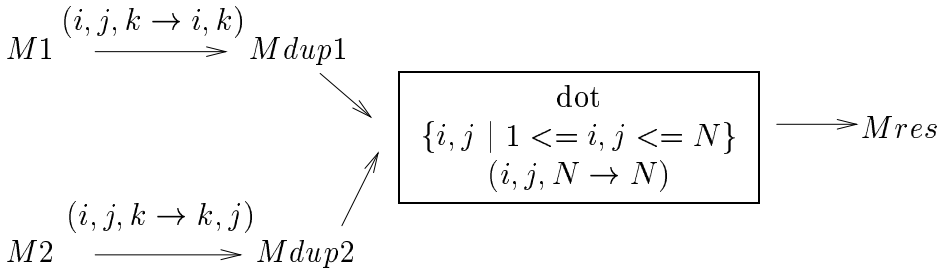


Figure 4: Structured dependence graph of the Matrix product `MM` of program 8

schedule already computed, and the schedules obtained are often more easily synthesized (for instance, we can easily obtain a pipe-line between successive use of the same system, see [26]). On the other hand, such structured linear schedules do not always exists and, even if they do, it is sometimes very difficult to find them. In this section we depict two cases in which these structured schedules can be found.

The problem is the following: given an Alpha structured program, in which cases can we find a linear structured schedule for this structured program? Of course a necessary condition is that the inlined system admits a linear schedule but this is not sufficient. We are interested in detecting such situations statically (i.e. without actually performing the scheduling).

We describe two types of structured programs where structured scheduling exists, and for each case, we propose an algorithm for finding linear structured scheduling using a linear schedule tool similar to the one described in section 4.

5.1 The SDG has no cycle labeled with a system name

The simplest case where we can find a linear structured scheduling is the case where the structured dependence graph does not contain any cycle in which one edge is labeled by a system name. For instance, any $N \times N$ matrix product can be expressed as N^2 dot products. On program 8 is represented the Alpha structured program for an $N \times N$ matrix product, the corresponding structured dependence graph is represented on figure 4. Such a specification corresponds to n^2 parallel use of the system `dot`. A natural method for obtaining structured scheduling for this kind of structured programs is to schedule the `dot` system first, and then to deduce constraints for the scheduling of the system `MM`.

Here, for instance, the scheduling of system `dot` would give:

$$T_{V1}(k) = 0 \quad T_{V2}(k) = 0 \quad T_{Acc}(k) = k \quad T_{res} = 1 + N \quad . \tag{4}$$

There are N^2 such `dot`, each instance of `dot` is labeled by a unique couple (i, j) in the extension domain: $\{i, j \mid 1 <= i, j <= N\}$. For a given `dot` product instance (say instance

```
system dot : { N | N >= 2}
               (V1,V2 : {k | 1<=k<=N} of integer )
       returns (res:  integer);
var
  Acc : {k | 0<= k <= N } of integer;
let
  Acc[k] =
     case
        {| k = 0 } : 0[];
        {| k > 0 } : Acc[k-1]+V1[k]*V2[k];
     esac;
  res[] = Acc[N];
 tel;

system MM : { N | N >= 2}
               (M1,M2 : {i,j | 1<= i,j <= N} of integer)
       returns (Mres : {i,j | 1<= i,j <= N} of integer);
var
  Mdup1,Mdup2: {k,i,j | 1<= i,j,k <= N} of integer;
let
   Mdup1[k,i,j]=M1[i,k];
   Mdup2[k,i,j]=M2[k,j];
   use {i,j| 1<= i,j <= N} dot[N] (Mdup1,Mdup2)
                        returns (Mres);
tel
```

Program **8:** Structured Alpha specification of the matrix product using N^2 dot products

(i_1, j_1)), scheduling this instance at time t means exactly that the inputs `Mdup1`$[i_1,j_1,k]$ and `Mdup2`$[i_1,j_1,k]$, $1 <= k <= N$ are available at time t and that, following from (4), the output `Mres`$[i_1,j_1]$ will be available at time $t + N + 1$. Here, the transition of the information between `dot` and `MM` is obvious since the `"N"` parameter of `dot` is the same as the `"N"` parameter of `MM` (the parameter assignment function is $(i, j, N \rightarrow N)$. In general, one should use the parameter assignment function (defined in the subsystem use) to bring back the schedule of system called on the caller variables. Here, if the parameter assignment function had been $(i, j, N \rightarrow f(i, j, N))$, the output `Mres`$[i_1,j_1]$ should be available at time $t + f(i, j, N) + 1$.

Hence we have proposed some constraints that must respect the schedule of system `MM` to make use of the schedule of the subsystem `dot`. We must impose these constraints by mean of linear constraints upon the indices, without knowing in advance the date t. We propose to do that in the following way:

- impose that the scheduling vectors of the variables `Mdup1,Mdup2` and `Mres` have the same (unknown) coefficient on i, j. The reason why we impose that is the following: if these coefficients are different, then two instances of use of `dot` will have their input data arriving at different dates, hence we cannot hope to use the same schedule for these two instances;
- set the coefficient of k to 0 for variables `Mdup1,Mdup2` (because the variables `v1[k]` and `v2[k]` of `dot` where supposed to be available at time 0);
- and set constraints on the remaining coefficients (i.e. coefficients on N and constant part) such that $T_{Mres}(i,j) - T_{Mdup1}(i,j,k) = N + 1$.

The constraint of the first point will always be set in the rest of the paper, it is fundamental in structured scheduling. This constraint is called the *Extension domain constraint*. One convenient way to ensure the above constraints is to impose the following constraints on the coefficients of timing vectors:

$$\begin{array}{ll} \tau_{Mdup1,i} = \tau_{Mdup2,i} = \tau_{Mres,i} & \tau_{Mres,N} - \tau_{Mdup1,N} = \tau_{Mres,N} - \tau_{Mdup2,N} = 1 \\ \tau_{Mdup1,j} = \tau_{Mdup2,j} = \tau_{Mres,j} & \alpha_{Mres} - \alpha_{Mdup1} = \alpha_{Mres} - \alpha_{Mdup2} = 1 \\ \tau_{Mdup1,k} = \tau_{Mdup2,k} = 0 & \end{array}$$

The fact of imposing these constraints (instead of expressing the exact constraints with Farkas Lemma) does not reduce much the space of solution, it corresponds to the assumption that the constraints are valid on the whole space (and not only on the domains of the variables). This allow not to use Farkas lemma to impose these constraints and the resulting schedule is in general equivalent to the one obtain with the exact constraints.

The method exemplified here will always work when the structured dependence graph does not contain any cycle with an edge labeled by a system name (and provided that each system used has a linear schedule). The general procedure for scheduling a structured program is a little bit more complicated as we must introduce additional constraints, from inputs for instance. We propose the following algorithm for scheduling an Alpha structured program `l1` with this property:

Algorithm A

`for` *each system call* `do`
    compute the extension domain constraints for actual inputs and outputs
`enddo`
perform a topological sort of the structured dependence graph of `l1`
Partition the graph into subgraph, each component containing
        one sub-system call, the component graph being acyclic
`for` *each component (from inputs to outputs)* `do`
    `If` *component contains a system call (to* `s1`*)* `do`
        compute the constraints on actual inputs of `s1`

```
        schedule the sub-system s1 (with constraints on formal inputs)
        deduce the schedules of the actual inputs and outputs of s1
    enddo
    schedule the variable of l1 of the current component with
        extension domain constraints and constraints deduced for I/O of s1
enddo
```

Note that this algorithm can be applied even if the schedule of the a system used `s1` has already been computed. In that case, instead of computing the schedule of `s1`, one just have to *shift* it in order to fit with the constraints on actual inputs of `s1`.

When the structured dependence graph contains a cycle with a system call, it often leads to multi-dimensional scheduling (non linear). We will study how to obtain structured multi-dimensional scheduling in section 6. Here, we isolate a property which ensure the existence of a linear schedule even if the structured dependence graph contains a cycle with a system use.

5.2 If the structured program is reducible

The content of this section was inspired by [26] where a particular program transformation called *bit level refinement* is introduced. This transformation corresponds to a refinement of an Alpha specification, from word level to bit level. The programs obtained after this transformation are called *reducible* and are such that they have a structured scheduling.

We first introduce formally the notion of reducibility, then we isolate a sufficient condition for a reducible structured program to have a structured linear schedule and we propose an algorithm to find it.

Definition of a reducible structured program

Given an Alpha structured program, a given system use involves some input and output variables. We will define a property of the call (called *separated dependencies property*), that basically indicates that the use is similar to a pointwise operator.

Definition 5.1 *(*separated dependencies property*) Given a use, in an Alpha structured program* `l1`*, of a system* `s1` *with an extension domain of dimension* k*. Let* $G_{\texttt{s1}} = (V_{\texttt{s1}}, E_{\texttt{s1}})$ *be the strongly connected component of the structured dependence graph containing the actual input and output variables of* `s1`*. This system call has the separated dependencies property if and only if, for each dependency of* $E_{\texttt{s1}}$ *from a variable* `A` *with* n *indices to a variable* `B` *with* m *indices, the image of the* $n-k$ *last indices of* `A` *is exactly the* $m-k$ *indices of* `B` *(i.e the dependency is identity on its last* $n-k$ *indices).*

The separated dependencies property gives the opportunity to consider two types of indices, the *caller* indices (k last indices: extension indices on which dependency appear only in the caller `l1`) and the *callee* indices (indices on which dependency happen only inside the called system).

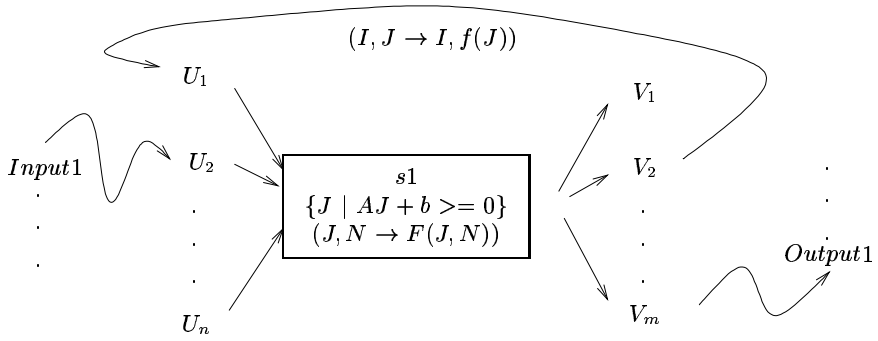


Figure 5: Generic SDG of a reducible structured program `l1` with only one system call (to `s1`). I represent the callee indices and J represents the caller indices.

Definition 5.2 *An Alpha structured program is* **reducible**, *if and only if,*

- *each system call has separated dependencies property,*
- *each strongly connected component of the* SDG *contains at most one system call.*

An example of a reducible structured program is given in section 5.3 (page 26) From a reducible structured program, it is possible to build a new system where the callee indices have disappeared and the `use` are replaced by special pointwise operators. This transformation corresponds to the reverse transformation of the refinement performed in [26]. We will not insist on this transformation but we will show that under certain condition, a reducible structured program admits a linear structured scheduling.

Schedule of a reducible structured program

Here we give a possible algorithm for scheduling reducible structured programs. For the sake of simplicity, we explain the scheduling process by focusing on one single cycle. The process is easily extensible and is exemplified in section 5.3.

The second property of the definition of a reducible structured program allows the SDG of the program to be decomposed into strongly connected components (each one containing only one system call) and schedule them successively. Hence, here we will only consider a simple Alpha structured program with only one system call. The general form of the SDG of such a reducible structured program `l1` is represented in figure 5. On this figure we have represented only one cycle in the dependence graph ($U_1 \to V_2 \to U_1$), but there may be many, each of which having the separated dependencies property. Also, the dependence path from V_2 to U_1 may be longer than one but the resulting dependency will always have the form $(I, J \to I, f(J))$ because of the separate dependencies property.

Here we make a strong assumption: we suppose that we have scheduled the sub-system `s1` and that the schedules T^{s1} obtained for the formal argument corresponding to U_1 and

O_2 are:

$$\begin{array}{l} T^{s1}_{U_1}(I) = \tau^{s1} I + \alpha^{s1} \\ T^{s1}_{V_2}(I) = \tau^{s1} I + \alpha^{s1} + c \quad , \end{array} \tag{5}$$

where c is a constant value (hence, the two schedules differ only by a constant). We will try to find a schedule T^{l1} of structured program `l1` by forgetting the call to `s1` and replacing the corresponding constraints by the constraints (6):

$$\begin{array}{l} \tau^{l1}_{U_1,I} = \tau^{s1} \\ \tau^{l1}_{V_2,I} = \tau^{s1} \\ \alpha^{l1}_{V_2} - \alpha^{l1}_{U_1} = c \\ \tau^{l1}_{U_1,J} = \tau^{l1}_{V_2,J} \\ T^{l1}_{U_1}(I,J) - T^{l1}_{V_2}(I, f(J)) \geq 1 \quad . \end{array} \tag{6}$$

The three first constraints correspond to the constraints inherited from the system call, the fourth constraints correspond to the fact that variable belonging to the same system call should have the same coefficient on j (extension domain constraints) and the last constraint is for the dependency from V_2 to U_1.

We will obtain schedule T^{l1} for U_1 and V_2, which will be valid for `l1`. Indeed, the dependence $U_1(I,J) \rightarrow V_2(I,J)$ (which must have delay c) is ensured because $T_{V_2}(I,J) - T_{U_1}(I,J) = c$ (first 4 constraints of (6)) and the dependence $V_2(I, f(J)) \rightarrow U_1(I,J)$ is ensured because of the last constraint of (6).

The process described above allows one to find structured scheduling in a structured manner (i.e without scheduling the inlined program, but only scheduling system by system) as soon as condition (5) is respected (i.e that in every system call, the delay between input and output is constant). We do not really have an immediate process to determine this condition (equation (5)), but it can be checked by performing the scheduling of the system used `s1`, adding the constraints that linear part of the concerned inputs (U_1) and outputs (V_2) are egal.

The question that remains is: is this condition necessary? It is clear that, if $T_{U_1}(I) - T_{V_2}(I)$ is a function $g(I)$ that really depends on I, then the global schedule of `l1` would look like $h(J) * g(I)$ which is not linear*. Hence, the necessary condition would be that the exact computation time of `V2[i]` and `U1[i]` differ only by a constant. However, it may happen for some degenerated domains of variables, that the condition (5) is not respected and still the difference $T_{U_1}(I) - T_{V_2}(I)$ is constant, but these cases are very rare and do not occur in practice. Hence the condition we give in (5) is a valid candidate for selecting program which have a structured linear schedule.

We have explained how to set the constraints in order to schedule a reducible structured program. The complete algorithm is basically the algorithm A described in section 5.1 with the special procedure for setting the constraints described here. At present time, we do not have a systematic way of providing linear structured scheduling for structured programs

*Except if the f dependency function is identity, which is not forbidden but which corresponds to a very strange programming habit.

which are not in one of the cases we studied in section 5.1 and 5.2. In next section we apply this algorithm to a simple reducible program.

5.3 Example of structured linear scheduling of a reducible program

```
system Ex1 : { N,W | N,W >= 2}
               (a : {i,j | 0<= i <= W-1; 1<=j <= N} of boolean)
       returns (b : {j | 1<= j <= N} of boolean);
 var
Bin1,Bin2,Bout: {i,j | 0<= i <= W-1; 1<=j <= N} of boolean;
 let
  Bin1[i,j] = a[i,j];
  Bin2[i,j] = case
               {| j=1} : False[];
               {| j>1} : Bout[i,j-1];
             esac;
  use {j| 1<= j <= N} Plus[W] (Bin1,Bin2)
                      returns (Bout);
  b[j]=Bout[W-1,j];
tel
```

Program **9:** Reducible Alpha structured program `Ex1`, the `Plus` system adds two vectors of bits as if they were representing integers.

We have built a simple example for the sake of clarity, a more complex and more realistic example can be found in [26]. Consider the Alpha structured program `Ex1` of program 9 (the system `Plus` has not been represented). Its structured dependence graph is represented on figure 6, one can easily see that the call to `Plus` has the separated dependencies property (the strongly connected component of the graph being the sub-graph containing $Bin_1, Bin_2, Bout$), hence this strucured program is reducible.

We want to schedule `Plus` and obtain a schedule that respects condition (5), hence we will impose the constraints: $\tau^{Plus}_{Bin_1} = \tau^{Plus}_{Bin_2} = \tau^{Plus}_{Bout}$. The schedule obtained is (7). It has the required property (indeed, when we perform a classical addition on integers represented as vectors of W bits, the total execution time is W because of the carry transition, but each input data is used only once to produce the corresponding result).

$$\begin{array}{l} T^{Plus}_{Bin_1}(b) = b \\ T^{Plus}_{Bin_2}(b) = b \\ T^{Plus}_{Bout}(b) = 1 + b \end{array} \tag{7}$$

Hence, the constraints (5) is respected and we will be able to find a linear structured scheduling for system `Ex1`. To do that, as indicated previously, we schedule `Ex1`, we forget the

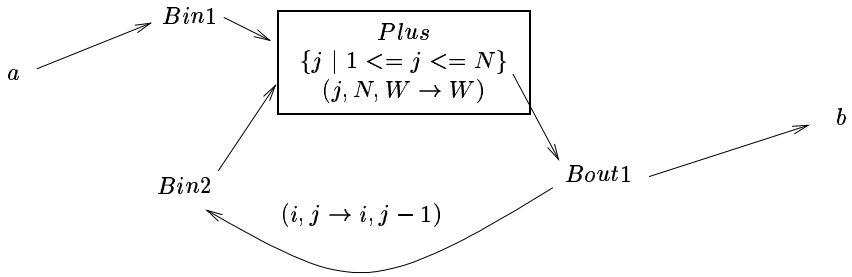


Figure 6: SDG of the system `Ex1` of program 9

call to `Plus` and we add the constraints:

$$\begin{array}{lll} \tau_{Bin_1,j} = \tau_{Bin_2,j} = \tau_{Bout,j} & & \tau_{Bin_1,N} = \tau_{Bin_2,N} = \tau_{Bout,N} \\ \tau_{Bin_1,i} = \tau_{Bin_2,i} = \tau_{Bout,i} = 1 & & \tau_{Bin_1,W} = \tau_{Bin_2,W} = \tau_{Bout,W} \\ \alpha_{Bout} - \alpha_{Bin_1} = 1 & \alpha_{Bout} - \alpha_{Bin_2} = 1 & \tau_{Bout,j} + \alpha_{Bin_2} - \alpha_{Bout} >= 1 \end{array}$$

The resulting schedule is:

$$\begin{array}{l} T_a^{Ex1}(i,j) = 0 \\ T_{Bin_1}^{Ex1}(i,j) = -1 + i + 2j \\ T_{Bin_2}^{Ex1}(i,j) = -1 + i + 2j \\ T_{Bout}^{Ex}(i,j) = i + 2j \\ T_b^{Ex1}(j) = 2j + W \quad . \end{array} \tag{8}$$

One can check that it is valid (each dependence is respected) and that it is structured (one can use the schedule (7) for the internal schedule of each use of the `Plus` system). This schedule is illustrated on figure 7 by a possible VLSI implementation.

6 Structured multi-dimensional scheduling

As we have seen in the previous section, it is hard to ensure that an Alpha structured program has a linear structured schedule. If we absolutely need a structured schedule, we must be able to compute structured non-linear schedules. In the cyclic scheduling community, non linear scheduling has been studied through *multi-dimensional* scheduling [1, 18, 30]. Before proposing some strategies for providing structured multi-dimensional schedules, we briefly explain how multi-dimensional schedule is computed and how it is interpreted in term of physical time. This is necessary to understand the objectives of structured multi-dimensional scheduling.

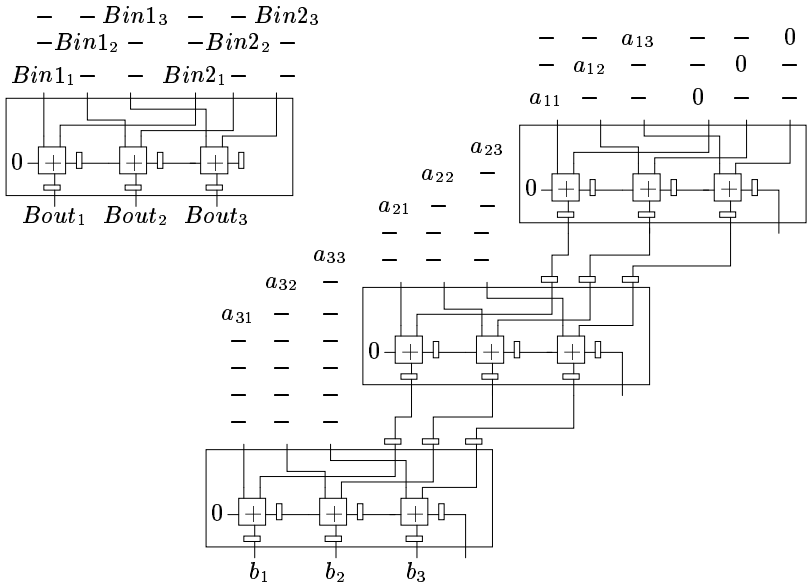


Figure 7: a possible VLSI interpretation for the structured scheduling (7) found for system `Plus` (on the left) and for the structured scheduling (8) of system `Ex1` (program 9) on the right. Note that each use of the system `Plus` is a straight copy of the circuit on the left. In that particular case, the three successive use on the `Plus` system could be realized by re-using the same hardware

6.1 Computing multi-dimensional schedules

The basic principles of multi-dimensional scheduling were settle down by Karp, Miller and Winograd. Detailed explanations can be found in [1, 18, 30]. The idea is the following. Given an Alpha system, if it does not have a linear schedule, it means that a linear expression upon indices is not sufficient to provide a partial order that respects all dependencies of the system. Hence we will look for a vector of linear expressions (upon the indices) as the *date* of execution. The precedence will be expressed as lexicographic precedence ($\ll$) on the components of this vector. For instance, the following multi-dimensional schedule is valid for the program 10:

$$T_a = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad T_{acc}(i,j) = \begin{pmatrix} j \\ i-1 \end{pmatrix} \quad T_b = \begin{pmatrix} M+1 \\ 0 \end{pmatrix} \tag{9}$$

This schedule has two dimensions, it does not really indicate an absolute date of computation for each variable but a relative order: `acc[i,j]` is computed before `acc[i-1,j+1]` because $T_{acc}(i,j) \ll T_{acc}(i-1,j+1)$, and similarly, `b[]` is computed after `acc[i,j]`. To figure out what represent these vectors dates, one can interpret the components of a multi-dimensional schedule vector as hours, minutes, seconds, etc. However this interpretation is realistic only if all the variables have the same rectangular domains, which is not always the case.

Computing multi-dimensional schedule is based on the following idea: if we cannot satisfy a dependency (for instance $T_{acc}(i,j) - T_{acc}(i-1,j) \geq 1$), we introduce an additional dimension in the timing function vector (T becomes $\begin{pmatrix} T^1 \\ T^2 \end{pmatrix}$), we try to satisfy the constraint $T^1_{acc}(i,j) - T^1_{acc}(i,j) \geq 0$ for the function T^1 (first component) and leave the work of *really satisfying* the dependency on the new dimension T^2: $T^2_{acc}(i,j) - T^2_{acc}(i,j) \geq 1$. The schedule showed in (9) respects both these constraints. Hence, computing multi-dimensional schedules can be done with a classical schedule tools like the one described in section 4, provided that we add a mechanism for handling the process described above. One way of handling this is explained in [18] (this method attempts to minimize the number of dimensions of the resulting schedule).

Even if the multi-dimensional scheduling provided great improvements in the power of analysis for parallelization techniques, it raised some problems. First, a multi-dimensional schedule does not directly give an operational semantic for an Alpha system, i.e. it is not obvious to deduce a valid order of operations from the expression of a multi-dimensional schedule (while this was obvious with linear scheduling as we simply had to interpret linear expressions as values of a global clock counter). Second, one easily see the problem when we try to build structured scheduling: in section 5, we used to search schedules by imposing constraints on the coefficients of these schedules. In the multi-dimensional case, we cannot know in advance, which dependency is satisfied at which level, it is impossible to enforce values for timing vectors, hence we must propose another method for finding structured scheduling.

6.2 Interpretation of multi-dimensional scheduling for VLSI

There are basically two ways of interpreting a multi-dimensional scheduling for a VLSI system. Either one considers that the system has only one clock (as classical systolic arrays) or one allows the system to have several clocks having relation between them (as often used in real time language [12]). We will describe how to realize each interpretation and illustrate it on a simple example. Consider the system of program 10, it corresponds to a sequential transition of a single data (`a[]`) in a N^2 matrix. It is obvious that `acc[N,1]` cannot be computed before time N, and recursively that `acc[N,j]` cannot be computed before time $j * N$, (which is not a linear expression). A valid multi-dimensional scheduling has been shown in (9).

```
system N2Acc : { N,M | N >= 2}
               (a : integer)
       returns (b :  integer);
var
 acc: {i,j | 1<=i<=N; 1<=j<=M} of integer;
let
 acc[i,j] = case
             {| i=1;j=1} :a[];
             {| 1<i<=N} :acc[i-1,j]+1[];
             {| j>1; i = 1}: acc[N,j-1]+1[];
           esac;
 b[]= acc[N,N];
tel
```

Program **10:** Alpha structured program for sequential accumulation in a matrix

From this schedule we propose two ways of finding time for each computation.

Single clock

We consider that our VLSI system has a single clock, hence we must express the schedule given by the time vector in term of the counter of clock ticks. We will propose a non linear expression corresponding to the multi-dimensional time vector. The expression is built with algorithm B.

Algorithm B

Consider a multi-dimensional schedule T of dimension k for an Alpha program with n variables (the variables of the program are called $A_1, \ldots, A_n$).
`for` *each component x of the time function vector T* `do`

`for` *each variable of the program* A_y `do`
Compute the range $(Min_{x,y}, Max_{x,y})$ *of the linear expression*
$T^x_{A_y}(I)$ *when* I *varies in the domain of* A_y
`enddo`
Compute $Min_x = \min_y(Min_{x,y})$ *and* $Max_x = \max_y(Max_{x,y})$
call $r_x = Max_x - Min_x + 1$
`enddo`
the value for time $T(I)$ *is:*
$val(T(I)) = T^k(I) + r_k T^{k-1}(I) + \ldots + \Pi_{l=2}^k r_l \; T^1(I)$

the validity of the schedule $val \circ T$ obtained by algorithm B is ensured if and only if:

$$T_A(I) \ll T_B(J) \Rightarrow val(T_A(I)) < val(T_B(J)) \quad .$$

This property is ensured by our algorithm. Indeed, suppose $T_A(I) \ll T_B(J)$, that is, there exists $h,\ 0 \leq h \leq k$ such that $T^x_A(I) = T^x_B(J)$ for $1 \leq x < h$ and $T^h_B(J) - T^h_A(I) \geq 1$.

$$\begin{array}{l} val(T_B(J)) - val(T_A(I)) = \\ \Pi_{l=h+1}^k r_l (T^h_B(J) - T^h_A(I)) + \ldots + T^k_B(J) - T^k_A(I) \geq \\ \Pi_{l=h+1}^k r_l + \Pi_{l=h+2}^k r_l (T^{h+1}_B(J) - T^{h+1}_A(I)) + \ldots + T^k_B(J) - T^k_A(I) \quad . \end{array}$$

By definition of r_{h+1}, we have: $r_{h+1} + T^{h+1}_B(J) - T^{h+1}_A(I) \geq 1$, thus we obtain:

$$\begin{array}{l} val(T_B(J)) - val(T_A(I)) \geq \\ \Pi_{l=h+1}^k r_l + \Pi_{l=h+2}^k r_l (1 - r_{h+1}) + \ldots + T^k_B(J) - T^k_A(I) = \\ \Pi_{l=h+2}^k r_l + \Pi_{l=h+3}^k r_l (T^{h+2}_B(J) - T^{h+2}_A(I)) + \ldots + T^k_B(J) - T^k_A(I). \end{array}$$

We can recursively suppress all term containing components of T_A and T_B in the right hand side and get to $val(T_B(J)) - val(T_A(I)) \geq 1$, hence the result.

In our example, $Min_2 = 0$ and $Max_2 = N - 1$ (maximum value taken by one of the expressions: 0 and $i - 1$ when i varies in $\{i, j \mid 1 \leq i \leq N; 1 \leq j \leq M\}$), hence $r_2 = N$ and for instance schedule $T_{acc}(i,j) = \begin{pmatrix} j \\ i-1 \end{pmatrix}$ corresponds to the date $Nj + i - 1$

Multiple clock

The other way of using a multi-dimensional schedule is to consider that we can build multi-rate circuits. There will be one clock by dimension of the schedule, the relations between the different clocks can be set (for instance) by using the ranges r_l computed in algorithm B: one top of c_l occurs every r_{l+1} top of c_{l+1}. Here, the logical time for time vector $T(I) = \begin{pmatrix} T^1(I) \\ T^2(I) \\ \ldots \\ T^k(I) \end{pmatrix}$ is itself, i.e $T^1(I)$th clock tick of clock c_1, ..., $T^k(I)$th clock tick of clock c_k.

Again, the data dependency is ensured because of the way we have built the clock relation (demonstration similar to the single clock case). This approach is more interesting in a VLSI synthesis context because the non linear expression obtained above is non trivial to implement on a VLSI circuit. In our example, the system of program 10 will lead to a circuit with two clocks (one being N times less frequent then the other), and the date of each computations on these clocks is indicated by (9).

6.3 Multi-dimensional schedule and structuring

As mentioned previously, the way multi-dimensional schedules are built is highly incompatible with the approach presented in section 5. At present time, the easiest way to obtain structured multi-dimensional schedule is to adopt an approach similar to the *instantaneous function* in language Signal [12], i.e consider that each call to a function is instantaneous, that the inputs and outputs are available at the same time and systematically force the addition of a dimension for each level of system called. For instance, in the program 11 (which can be considered as a structured version of program 10), the schedule of system `oneRow` is:

$$T_a^{oneRow} = 0 \quad T_{acc}^{oneRow}(i) = i \quad T_{res}^{oneRow} = N + 1 \quad . \tag{10}$$

```
                                  system NRow:{ N | N >= 2}
                                              (a : integer)
system oneRow : { N | N >= 2}         returns (b :  integer);
               (a : integer)      var
       returns (res: integer);     accTemp1:{j|1<=j<=N} of integer;
var                                accTemp2:{j|1<=j<=N} of integer;
acc:{i| 1<=i<=N} of integer;      let
 let                               accTemp1[j]=case
acc[i]=case                                  {| j=1}:a[];
        {|i=1}:a[];                          {| j>1}:accTemp2[j-1];
        {|>1}:acc[i-1];                     esac;
       esac;                      use {j|1<=j<=N} oneRow[N] (accTemp1)
res[]=acc[N];                              returns (accTemp2);
tel;                               b[]= accTemp2[N];
                                  tel;
```

Program **11:** Structured Alpha structured program without linear schedule

the schedule of system `NRow` can be done by forgetting the use and adding the following constraints:

$$\tau_{accTem1,j}^{NRow} = \tau_{accTemp2,j}^{NRow} \qquad \alpha_{accTemp1}^{NRow} = \alpha_{accTemp1}^{NRow}$$
$$\tau_{accTem1,N}^{NRow} = \tau_{accTemp2,N}^{NRow}$$

which gives:

$$T_a = 0 \quad T_{accTemp1}(j) = j \quad T_{accTemp2}(j) = j \quad T_b = 1 + N \quad . \tag{11}$$

And from (10) and (11) we derive the multi-dimensional timing function:

$$T_a = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad T_{accTemp1}(j) = \begin{pmatrix} j \\ 0 \end{pmatrix} \quad T_{accTemp2}(j) = \begin{pmatrix} j \\ N+1 \end{pmatrix} \quad T_b = \begin{pmatrix} N+1 \\ 0 \end{pmatrix}$$

This method can always be applied since in Alpha the system graph (graph of the calls between the systems of the structured program) is acyclic: two systems cannot call each others. However, there are several problems with this approach. First, as for the linear case depicted in section 5, this will not work on all Alpha program which admit multi-dimensional schedule. Again, this is due to the shortcut in the SDG for dependencies inside a system call: some output may not depend on some input, hence a cycle in the SDG may not be a real cycle. In particular, as in the linear case, if the dependency occurring from an output to an input (`accTemp2` $\rightarrow$ `accTemp1` in program 11) is identity, this method will not work. Currently we do not have a solution to this problem.

Another problem with this approach is that it forbids the possible pipelines between several `use` of the same (or different) systems. The output of a system will not be used before the system is completely executed, this is a great drawback in VLSI synthesis since pipeline is very often used. Hence, we would like to apply, as often as possible the algorithm A described in section 5.1, and when it appears that one strongly connected component cannot be scheduled in a linear way, then apply the method described above and add a dimension.

Let us formalize this technique, given any Alpha system, algorithm C will try to find a structured multidimensional scheduling, with as many as possible linear structured schedules for system calls.

Algorithm C

Perform a topological sort of the structured dependence graph of `l1`
`for` *each component* c `do`
    determine the constraints of the input to the component
    get schedule T_c *of this component by* **Algorithm A**
    `If` *it fails* `do`
        `for` *each system call (to* s_i*) of the component* `do`
            schedule s_i *(***Algorithm C***) and get a* k_i *dimensional schedule* T_c^i
        `enddo`
        get a k *dimensional schedule* T'_c *of the component* c *(forgetting the use)*
            with the constraints that input and output of a system use
            have exactly the same schedule
        get the schedule T_c *of the component by joining time vectors* T'_c
            and T_c^i *for each variables.*
    `enddo`
`enddo`

We still have to prove that the resulting schedule is valid, i.e. that all dependencies are respected. We have already proved that algorithm A provided linear schedules which respect dependencies. Remains to prove that the schedule found by our procedure (if algorithm A fails) respects dependencies inside a strongly connected component and that dependencies between strongly connected components are ensured.

Inside a strongly connected component, dependencies between input and output of a system `s1` are respected because they have the same coefficients in T'_c and that the recursive use of algorithm C will provide coefficients of T^i_c that ensure dependencies (a recurrence hypothesis should be set to prove that correctly, we assume that this property is clear enough not to prove it so formally). Inside the strongly connected component `c`, a dependency occurring in the `l1` system is ensured by T'_c (as T^i_c is appended at the end, it does not modify this). Dependency between strongly connected components are ensured because, before scheduling one component, we compute the constraints (on the first level of schedule) derived from the schedules of the all previous (in the topological order sense) components.

7 Structured scheduling for the singular value decomposition

In this section we show how we derived a structured scheduling for a realistic example with the method presented in this paper. The example chosen is a part of an algorithm for computing the singular value decomposition (SVD) of a $M \times 2N$ matrix. The SVD is used in many area (mobile telephone for instance [31]). Several method were proposed to find the SVD of a matrix (see [32] for a survey). One of these methods (explained in [33]) is a Jacobi-like algorithm with iterative applications of successive orthogonalization of pairs of columns in the matrix. This method has been specified in Alpha, in program 12, the `SWEEP` system represented one iteration of the Jacobi algorithm. `STEP` and the other systems called are represented in program 13, the system `STEP` corresponds to the orthogonalization of n particular pairs of columns (each column is involved one time) as $A\ M \times 2N$). This specification is inspired from Blas libraries [34]. From this specification, it seems clear that the `STEP` system should have a linear schedule as it consist of `N` independent orthogonalization (each having an $O(M)$ complexity).

In this section, we adopt the convention that one equation takes one top, as explained in section 4, this can be modified and greatly influence the resulting scheduling. Following the method explained in section 6, we start by the scheduling of system `SWEEP` and we suppose, for instance, that coefficient $A_{i,j}$ of the input matrix `A` is available at step i (constraint from a previous computation): $T_A(i,j) = i$. The SDG of sweep contains only one strongly connected component which includes all local variables: `C1,C2,C1r,C2r`. The first thing to compute are the constraints on `C1,C2` deriving from the input `A`. Here, we must have: $T_{C1}(i,j) \geq i+1$ and $T_{C2}(i,j) \geq i+1$. Now, we go down to the schedule of `STEP` with the constraints on formal inputs `C1,C2`; $T_{C1}(i,j) \geq i+1$ and $T_{C2}(i,j) \geq i+1$

```
system SWEEP : {M,N | 4<=M; 4<=N}
               (A : {i,j | 1<=i<=M; 1<=j<=2N} of real)
       returns (B : {i,j,l | 1<=i<=M; 1<=j<=N; 0<=l<=1} of real);
var
  C1 : {i,j,k | 1<=i<=M; 1<=j<=N; 0<=k<=2N-1} of real;
  C2 : {i,j,k | 1<=i<=M; 1<=j<=N; 0<=k<=2N-1} of real;
  C1r : {i,j,k | 1<=i<=M; 1<=j<=N; 1<=k<=2N-1} of real;
  C2r : {i,j,k | 1<=i<=M; 1<=j<=N; 1<=k<=2N-1} of real;
let
  use {k | 1<=k<=2N-1} STEP[M,N]
        (C1.(i,j,k->i,j,k-1), C2.(i,j,k->i,j,k-1))
         returns  (C1r, C2r) ;
  C1[i,j,k] =
      case
        {| k=0} : A[i,2j-1];
        {| 1<=k} :
           case
             {| j=1} : C1r[i,j,k];
             {| j=2} : C2r[i,1,k];
             {| 3<=j} : C1r[i,j-1,k];
           esac;
      esac;
  C2[i,j,k] =
      case
        {| k=0} : A[i,2j];
        {| 1<=k} :
           case
             {| j=N} : C1r[i,j,k];
             {| j<=N-1} : C2r[i,j+1,k];
           esac;
      esac;
  B[i,j,l] =
      case
        {| l=0} : C1[i,j,2N-1];
        {| l=1} : C2[i,j,2N-1];
      esac;
tel;
```

Program **12:** System SWEEP performing one step of the (one-sided) Jacobi Algorithm for the SVD

Here, we can notice that the SDG of STEP is acyclic, as are all the SDG of the systems called by STEP. Hence, we will be able to find a linear scheduling using algorithm A (provided that each system has a linear scheduling). We transmit to the schedule of DROT the constraints

on formal input `v1,v2`: $T_{v1}(i) \geq i+1$ and $T_{v2}(i) \geq i+1$. `DROT` uses the two subsystems `DDOT` and `DNRM2`. The schedule of `DNRM2` with the constraint of formal input `v`: $T_v(i) >= i+1$ gives:

$$T_v(i) = 1+i \quad T_{vcumul}(i) = 2+i \quad T_{norm} = 3+M \quad . \tag{12}$$

The schedule of `DDOT` is similar:

$$T_{v1}(i) = 1+i \quad T_{v2}(i) = 1+i \quad T_{vcumul}(i) = 2+i \quad T_{pdtscal} = 3+M \quad .$$

Then we try to schedule system `DROT` with additional constraints on inputs `v1,v2` ($T_{v1}(i) = i+1$ and $T_{v2}(i) = i+1$) and local variables `alpha,beta,gama` ($T_{alpha} = 3+M$, $T_{beta} = 3+M$, $T_{beta} = 3+M$), the result is:

$$\begin{array}{llll} T_{v1}(i) = 1+i & T_{v2}(i) = 1+i & & \\ T_{alpha} = 3+M & T_{beta} = 3+M & T_{gama} = 3+M & \\ T_{eps} = 4+M & T_t = 5+M & T_c = 6+M & T_s = 7+M \\ T_{C1}(i) = 8+M & T_{C2}(i) = 8+M & & \end{array} \tag{13}$$

Now we are able to schedule `STEP`. Here, the `use` has an extension domain, hence we will add to the constraints deduced from (13), the so-called extension domain constraints: $\tau_{C1,j} = \tau_{C2,j} = \tau_{C1r,j} = \tau_{C2r,j}$, the resulting schedule is trivial here, and meet the linearity requirement mentioned previously:

$$T_{C1}(i,j) = T_{C2}(i,j) = 1+i \quad T_{C1r}(i,j) = 8+M \quad T_{C2r}(i,j) = 8+M \quad . \tag{14}$$

We finally come to the schedule of the `SWEEP` system. Attempting to find a linear schedule for this system will fail as the call to `STEP` does not have the separated dependencies property (in fact, `SWEEP` does not have a linear schedule). Hence, we will add a dimension for the call to `STEP` and schedule `SWEEP` imposing that, for each instance `k` of `STEP`, inputs and outputs are computed simultaneously. Hence the constraints:

$$\begin{array}{c} \tau_{C1,k} = \tau_{C2,k} = \tau_{C1r,k} = \tau_{C2r,k} \\ T_{C1}(i,j,k-1) = T_{C2}(i,j,k-1) = T_{C1r}(i,j,k) = T_{C2r}(i,j,k) \quad . \end{array}$$

Note that we can forget the input constraints coming from `A` because these have already been included in the schedule of `C1,C2` and do not influence directly other variables. The resulting linear schedule is:

$$\begin{array}{lll} T_A(i,j) = 0 & T_{C1}(i,j,k) = 1+k & T_{C2}(i,j,k) = 1+k \\ T_{C1r}(i,j,k) = k & T_{C2r}(i,j,k) = k & T_B(i,j,l) = 1+2N \end{array} \tag{15}$$

Schedule (15) and (14) finally give as multi-dimensional scheduling for system `SWEEP`:

$$\begin{array}{ll} T_A(i,j) = \begin{pmatrix} 0 \\ i \end{pmatrix} & T_B(i,j,l) = \begin{pmatrix} 1+2N \\ 0 \end{pmatrix} \\ T_{C1}(i,j,k) = \begin{pmatrix} 1+k \\ i+1 \end{pmatrix} & T_{C2}(i,j,k) = \begin{pmatrix} 1+k \\ i+1 \end{pmatrix} \\ T_{C1r}(i,j,k) = \begin{pmatrix} k \\ M+8 \end{pmatrix} & T_{C2r}(i,j,k) = \begin{pmatrix} k \\ M+8 \end{pmatrix} \end{array} \tag{16}$$

```
system DNRM2 : {M|4<=M}
      (v:{i|1<=i<=M} of real)
       returns (norm:real);
var
  vcumul:{i|0<=i<=M} of real;
let
  vcumul[i]=
    case
      {|i=0} : 0[];
      {|1<=i} : vcumul[i-1] +
                  v[i] * v[i];
    esac;
  norm[]=vcumul[M];
tel;


system DDOT :  {M|4<=M}
      (v1:{i|1<=i<=M} of real;
       v2:{i|1<=i<=M} of real)
       returns (pdtscal:real);
var
  vcumul:{i|0<=i<=M} of real;
let
 vcumul[i] =
  case
   {| i=0} : 0[];
   {| 1<=i} : vcumul[i-1] +
               v1[i] * v2[i];
  esac;
 pdtscal[] = vcumul[M];
tel;
```

```
system DROT:{M|4<=M}
     (v1:{i|1<=i<=M} of real;
      v2:{i|1<=i<=M} of real)
    returns
     (C1:{i|1<=i<=M} of real;
      C2:{i|1<=i<=M} of real);
var
 alpha,beta,gama,eps,t,c,s :real;
let
 use DNRM2[M] (v1)returns(alpha);
 use DNRM2[M] (v2)returns(beta);
 use DDOT[M] (v1,v2)returns(gama);
 eps[]=0.5*(beta-alpha)/gama;
 t[]=if (eps = 0) then 1 else
     if (eps > 0) then
     1/(eps+sqrt(1+eps*eps)) else
     -1/(-eps+sqrt(1+eps*eps));
  c[] = 1 / sqrt(1 + t * t);
  s[] = t * c;
  C1[i] = c[] * v1 - s[] * v2;
  C2[i] = s[] * v1 + c[] * v2;
 tel;

system STEP:{M,N | 4<=M; 4<=N}
(C1:{i,j|1<=i<=M; 1<=j<=N} of real;
 C2:{i,j|1<=i<=M; 1<=j<=N} of real)
returns
 (C1r:{i,j|1<=i<=M; 1<=j<=N} of real;
 C2r:{i,j|1<=i<=M; 1<=j<=N} of real);
let
 use {j | 1<=j<=N} DROT[M]
    (C1, C2) returns  (C1r, C2r) ;
tel;
```

Program **13:** system DROT (using DDOT AND DNRM2) computing the orthogonalization of two column of a matrix, is used by N times by system STEP which scans all the matrix.

One can check that this schedule is valid. It respects each dependence, for instance, $C1[i,j,k] \leftarrow C1r[i,j-1,k]$ is respected because:

$$\begin{pmatrix} k \\ M+8 \end{pmatrix} \ll \begin{pmatrix} 1+k \\ i+1 \end{pmatrix} \quad .$$

And the input constraints (from A) is also respected ($A[i,j]$ is scheduled at $\begin{pmatrix} 0 \\ i \end{pmatrix}$). Hence we obtained a valid structured multi-dimensional schedule for system SWEEP.

Another method for scheduling the computations of SWEEP would be to look for a non-structured scheduling, hence to inline all the structuring of SWEEP in order to obtain a single big system of recurrence equation and to schedule it with a classical multi-dimensional scheduling tool. The first problem with this approach is the computation time. In the present case, the LP generated from the unrolled SWEEP system is composed of 1081 variables and 1138 constraints, while the maximum size of LP generated with the previous method was for SWEEP: 196 variables and 226 constraints. Even if the structured scheduling method contains many manipulations, while this one consists in a single schedule, this complexity problem will become critical for bigger programs. The other problem is also crucial, as we cannot know in advance how many dimensions will compose the schedule, we must impose the constraints deriving from input A on the highest level: $T_A(i,j) >= i$, which will mean after scheduling: $T_A(i,j) \gg \begin{pmatrix} i \\ 0 \end{pmatrix}$, while the effective constraints in the first method was: $T_A(i,j) \gg \begin{pmatrix} 0 \\ i \end{pmatrix}$.

The resulting flat schedule is also bi-dimensional, it is shown in (17), were we only print the schedule relative to the original variables of SWEEP.

$$\begin{array}{ll} T_A(i,j) = \begin{pmatrix} -1+i \\ 0 \end{pmatrix} & T_B(i,j) = \begin{pmatrix} -8+M+16N \\ 0 \end{pmatrix} \\ T_{C1}(i,j) = \begin{pmatrix} 8k+M \\ 0 \end{pmatrix} & T_{C2}(i,j) = \begin{pmatrix} 8k+M \\ 0 \end{pmatrix} \\ T_{C1r}(i,j) = \begin{pmatrix} -1+8k+M \\ 0 \end{pmatrix} & T_{C2r}(i,j) = \begin{pmatrix} -1+8k+M \\ 0 \end{pmatrix} \end{array} \quad (17)$$

The second dimension is used for scheduling intermediate variables that come from the inlining of the systems used. Just note that the number *minutes* in one *hour* (i.e the number of time steps enumerated by the second dimension of the schedule) is roughly the same for schedule (16) (M+8) and schedule (17) (M) . The main differences between the two schedules come from the N parameter which appears in the final schedule of B (in (17)) and the $8k$ in the local variables (instead of k in (16)). The $8k$ coefficient comes from the fact that many dependencies are by default satisfied at the first level (while they were satisfied at lower level in the schedule (16)), the N come from the input constraint from A which is also ensured at the first level because of the reason mentioned above.

The flat schedule (17) increases the number of step of the scheduled SWEEP from $(2N+1)*(M+8) = O(MN)$ to $(16M+N-15)*M = O(M^2+NM)$. In our example, structured scheduling combines the advantages of preserving the structure of the program and providing a faster schedule.

8 Conclusion

We have presented a method for scheduling structured systems of recurrence equations. We first proposed to isolate cases where a structured SARE admits a linear structured scheduling. These cases can be identified without performing the whole scheduling process. In practice, these cases occur quite often, but as soon as the program has a quadratic time complexity, linear scheduling is not suitable. Hence we proposed an algorithm for finding multi-dimensional structured scheduling for structured SAREs. This algorithm allows to keep the structuring information and to simplify the scheduling complexity. It has been exemplified on a realistic example extracted form the ScalaPack library. A structured scheduling tool implementing this method can easily be built from a non-structured schedule tool. We also provided the features that was needed by a non-structured schedule tool if it is dedicated to VLSI synthesis.

This work have been done in order to provide a strategy for scheduling large Alpha systems in the MMAlpha environment. Many open questions remain. First, we do not know if it is possible to statically determine all the cases where a structured SARE admits a structured linear scheduling (neither for structured multi-dimensional scheduling of course). Algorithm C proposed for multi-dimensional structured scheduling is one possible method among many variants, it remains to evaluate this method and to compare it to other approaches. In particular, in this paper, we wanted to use a classical schedule tool as the basis of a structured scheduling tool. If we get rid of this implementation constraint, it may be possible to define completely different scheduling algorithms. Finally, we are certainly aware of the fact that the implementation of a scheduled structured SARE in hardware has still to be studied in depth, especially concerning the *real cost* of a multi-dimensional time.

References

[1] R.M. Karp, R.E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, July 1967.

[2] Leslie Lamport. The parallel execution of do loops. *Communications of The ACM*, 17(2):83–93, February 1974.

[3] P. Feautrier. Some efficient solutions to the affine scheduling problem, part I, one dimensional time. *Int. J. of Parallel Programming*, 21(5):313–348, October 1992.

[4] M. Wolf and M. Lam. Loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, Oct 1991.

[5] Alain Darte, Leonid Khachiyan, and Yves Robert. Linear scheduling is nearly optimal. *Parallel Processing Letters*, 1(2):73–81, 1991.

[6] H.T. Kung. Why systolic architectures? *Computer*, 15(1):37–46, 1982.

[7] P. Lee and Z.M. Kedem. Mapping nested loop algorithms into multidimensional systolic arrays. *IEEE Transaction On Parallel and Distributed System*, 1(1):64–76, January 90.

[8] D.I. Moldovan. On the analysis and synthesis of vlsi systolic arrays. *IEEE Transactions on Computers*, 31:1121–1126, 1982.

[9] Christophe Mauras, Patrice Quinton, Sanjay Rajopadhye, and Yannick Saouter. Scheduling affine parameterized recurrences by means of variable dependent timing functions. In S.Y Kung, Jr. E.E. Swartzlander, J.A.B. Fortes, and K.W. Przytula, editors, *Application Specific Array Processors*, pages 100–110. IEEE Computer Society Press, September 1990.

[10] E.A. Ashcroft and W.W. Wadge. Lucid, a formal system for writing and proving programs. *SIAM j. Comp.*, 3:336–354, 1976.

[11] P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. Lustre: a declarative language for programming synchronous systems. In *14th Symposium on Principles of Programming Languages*. ACM, Munich, Janvier 1987.

[12] P. LeGuernic, A. Benveniste, P. Bournai, and T. Gautier. SIGNAL: A data flow oriented language for signal processing. In *IEEE Workshop on VLSI*, 1984.

[13] M. Chen, Y. Choo, and J. Li. *Crystal: Theory and Pragmatics of Generating Efficient Parallel Code*, page Chapter 7. ACM Press, 1991.

[14] G.-R. Perrin, S. Genaud, and E. Violard. PEI: a theoretical framework for data-parallel programming. Technical report, ICPS, Strasbourg, 1994.

[15] C. Mauras. *Alpha: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones*. PhD thesis, Université de Rennes 1, IFSIC, December 1989.

[16] Yannick Saouter. *A propos de systèmes d'équations récurrentes*. PhD thesis, Université de Rennes 1, oct 1992.

[17] A. Darte. *Techniques de parallélisation automatique de nids de boucles*. PhD thesis, LIP ENS-Lyon, 1993.

[18] P. Feautrier. Some efficient solution to the affine scheduling problem, part II, multidimensional time. *Int. J. of Parallel Programming*, 21(6), December 1992.

[19] F. Dupont De Dinechin, P. Quinton, and T. Risset. Structuration of the alpha language. In W.K Giloi, S. Jahnichen, and B.D. Shriver, editors, *Massively Parallel Programming Models*, pages 18–24. IEEE Conmputer Society Press, 1995.

[20] Florent Dupont de Dinechin. *Systèmes structurés d'équations récurrentes : mise en œuvre dans le langage Alpha et applications*. Thèse de doctorat, université de Rennes I, January 1997.

[21] D. Wilde. A library for doing polyhedral operations. Technical Report 785, IRISA, Rennes, France, 1993.

[22] D. K. Wilde. The Alpha language. Research Report 999, IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France, Jan 1994.

[23] Florent de Dinechin and Sophie Robert. Hierarchical static analysis of structured systems of affine recurrence equations. In *Application Specific Array Processors*. IEEE Conmputer Society Press, August 1996.

[24] Florent De Dinechin and Patricia Le Moenner. Automatic synthesis of regular architectures optimized at the bit level. In *Workshop on Design Methodologies for Signal Processing*, Zakopane, Poland, August 1996.

[25] P. Le Moenner, L. Perraudeau, S. Rajopadhye, T. Risset, and P. Quinton. Generating regular arithmetic circuits with AlpHard. In *Massively Parallel Computing Systems (MPCS'96)*, May 1996.

[26] Florent De Dinechin. Libraries of schedule-free operators in alpha. In *ASAP*, 1997.

[27] Alain Darte and Yves Robert. Scheduling uniform loop nests. Technical Report 92-10, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, France, February 1992.

[28] Paul Feautrier and Nadia Tawbi. Résolution de systèmes d'inéquations linéaires; mode d'emploi du logiciel pip. Technical Report 90-2, Institut Blaise Pascal, UPMC, Laboratoire MASI, January 1990.

[29] Patrice Quinton and Yves Robert. *Systolic Algorithms and Architectures*. Prentice Hall and Masson, 1989.

[30] A. Darte and F. Vivien. Revisiting the decomposition of karp, miller and winograd. In *ASAP*, pages 13–25, 1997.

[31] Ed. F Deprettere, editor. *SVD and Signal Processing Algorithms, Applications and Architectures*, North-Holland, 1988. Elsevier Science Publishers B.V.

[32] G.H. Golub and C.H. Van Loan. *Matrix computations*. The Johns Hopkins University Press, 1989.

[33] R. P. Brent and F. T. Luk. The solution of the singular value and symmetric eigenvalue problems on multiprocessors arrays. *SIAM J. Sci. Statist. Comput*, 6:69–84, 1985.

[34] C.L. Lawson, R.J. Hanson, D. Kincaid, and F.T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Soft*, 5:308–323, 1979.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399